Let's add **print** and **definitions** and **calls** to our compiler

```
fn snek_print(val: i64) -> i64 {
  if (val == 1) {
    println!("false")
  } else if (val == 3) {
    println!("true")
  } else {
    println!("{}", val>>1)
  }
  return val;
}
```

(+ true 1)

mov rax,(true)

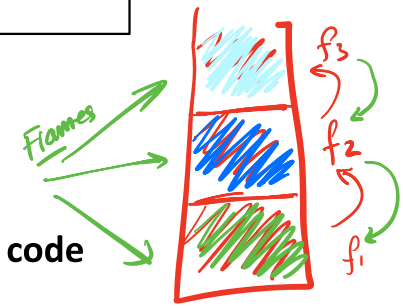
:

mov rdi, 33

(print 7)

rax=

```
enum Expr {
  ...
  Print(Box<e>),
}
```



Example

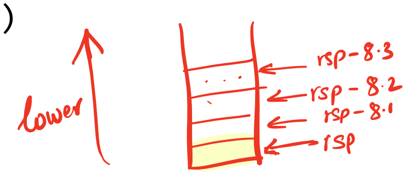
(print e)

<e>

"call-print"

Generated code

```
(let (x (print input))
  (+ x 1))
```



print

(print 7)

mov rdi, 7

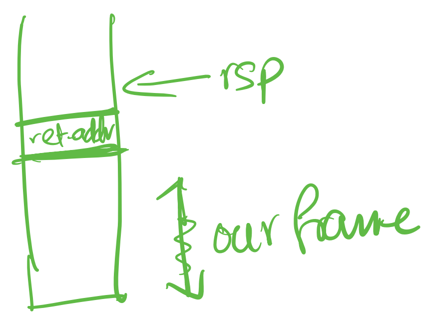
call snek_print

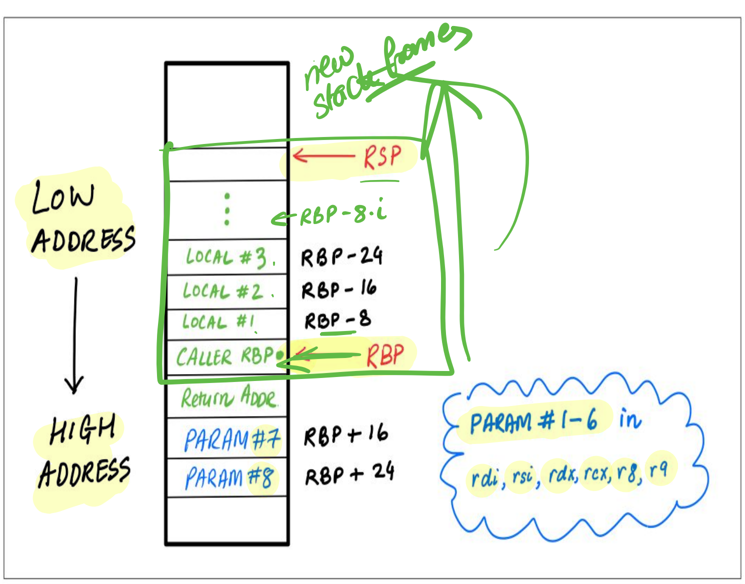
(print e)

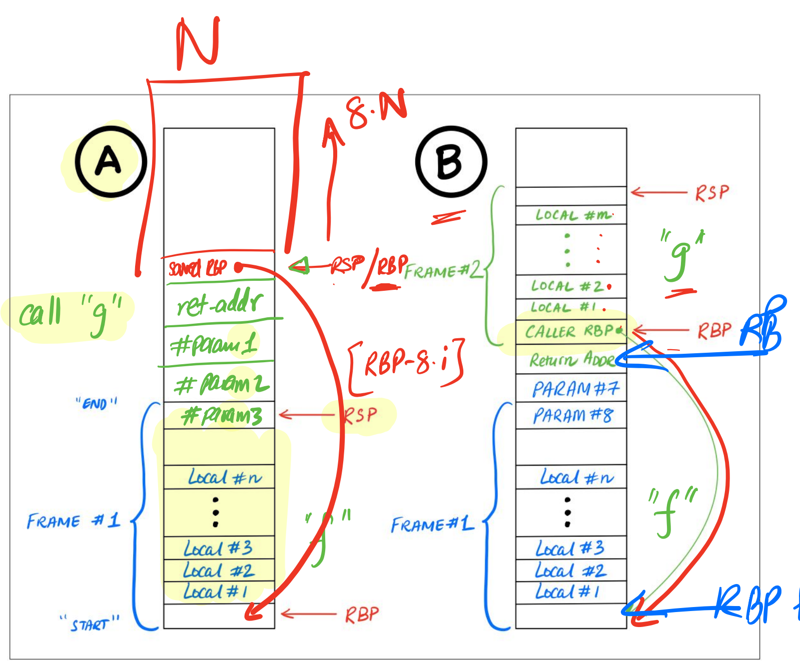
<e>

mov rdi, rax

call snek_print







Let's add **print** and **definitions** and **calls** to our compiler

```
<prog> := <defn>+ <expr>

<defn> := (defn (<name> <name>) <expr>)
        | (defn (<name> <name> <name>) <expr>)

<expr> := …
        | (<name> <expr>)
        | (<name> <expr> <expr>)
```

```
enum Defn {
  Fun1(String, String, Box<Expr>),
  Fun2(String, String, String, Box<Expr>),
}

enum Expr {
  ...
  Call1(String, Box<Expr>)
  Call2(String, Box<Expr>, Box<Expr>)
}
```

Example

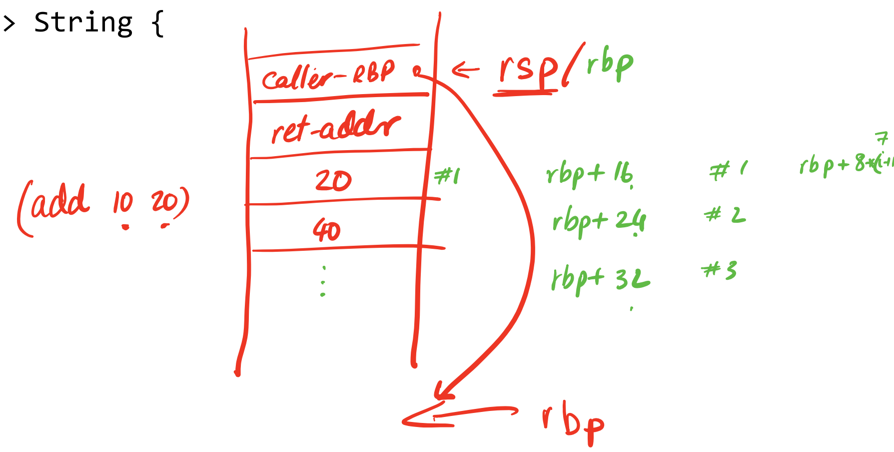
```
(defn (add x1 x2)
   (+ x1 x2))

(add input 10)
```

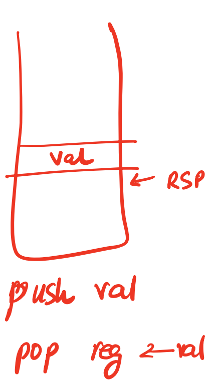
Generated code

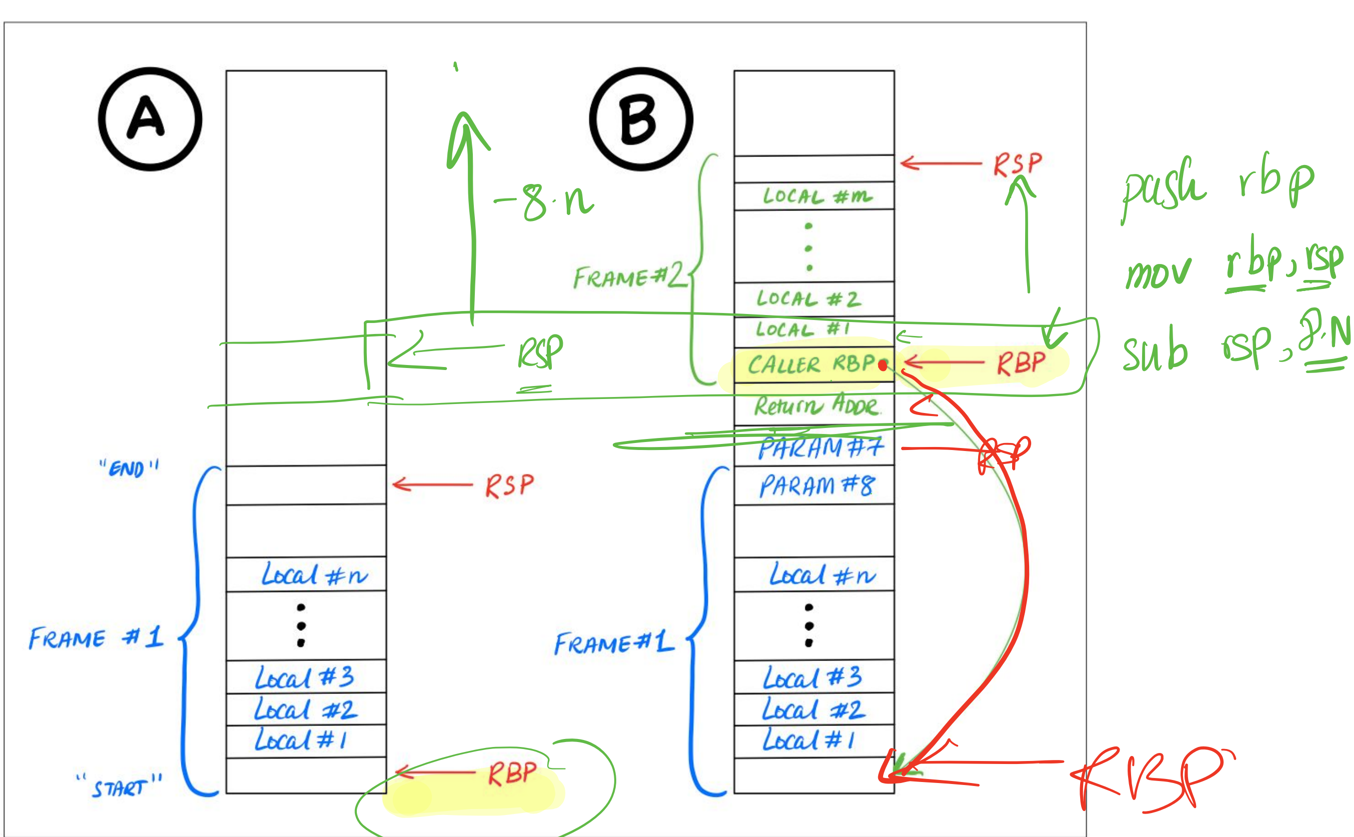
def

call

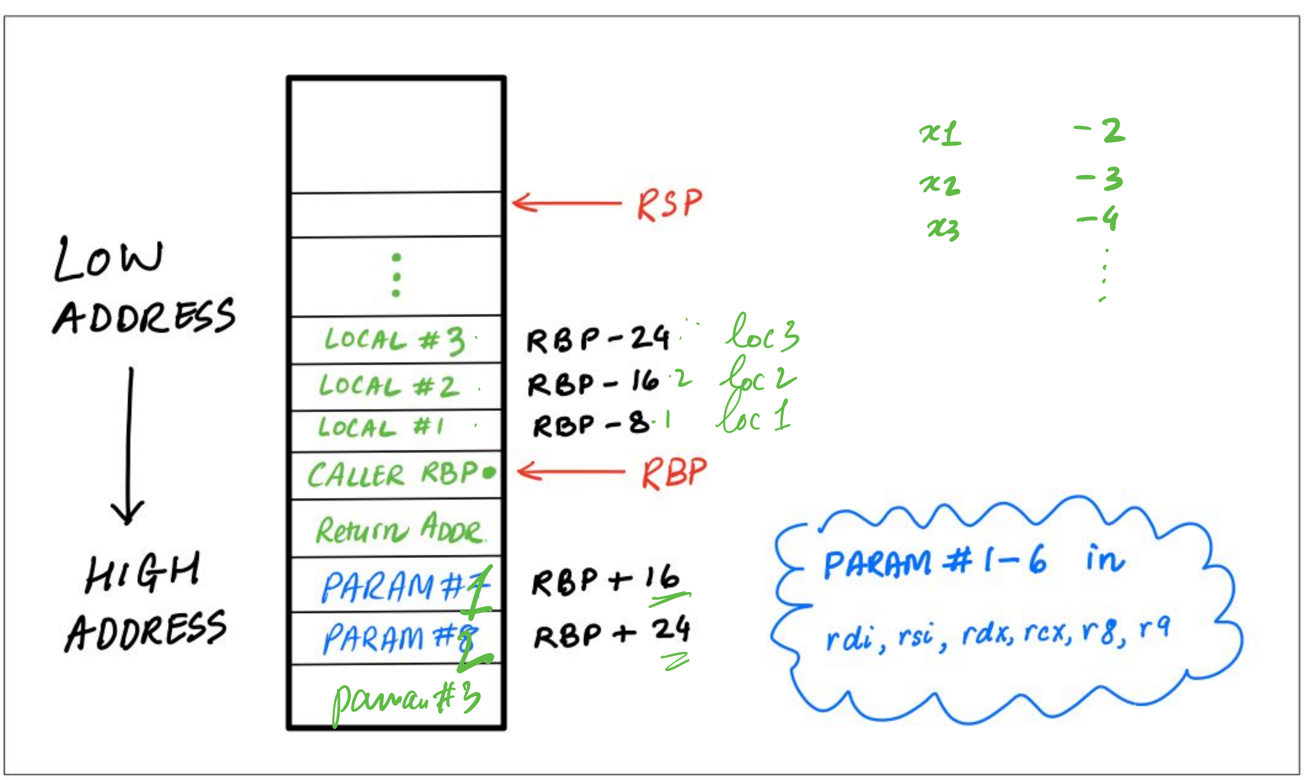
```
fn compile_def_body(args: &[String], sp: usize, body: &Expr, count: &mut i32) -> String
{
  let fun_entry = compile_entry(body, sp);
  let body_code = compile_expr(body, &init_env(args), sp, count, "time_to_exit");
  let fun_exit  = compile_exit();

  format!("{fun_entry}
          {body_code}
          {fun_exit}")
}

fn compile_entry(e: &Expr, sp: usize) -> String {
  let vars = expr_vars(e) + sp;
  format!("push rbp
           mov rbp, rsp
           sub rsp, 8*{vars}")
}

fn compile_exit() -> String {
  format!("mov rsp, rbp
           pop rbp
           ret")
}
```







```
fn compile_expr(...) -> String {
  match e {
    ...
    Expr::Call2(f, e1, e2) => {
      let e1_code = compile_expr(e1, env, sp, count, brk);
      let e2_code = compile_expr(e2, env, sp + 1, count, brk);
      format!("{e1_code}
              mov [rbp - 8*{sp}], rax
              {e2_code}
              push rax
              mov rcx, [rbp - 8*{sp}]
              push rcx
              call fun_start_{f}
              add rsp, 8*2")
      }
  }
}
```



Recursive Calls

```
sumTo(5)
==> 5 + sumTo(4)
        ^^^^^^^^
==> 5 + [4 + sumTo(3)]
             ^^^^^^^^
==> 5 + [4 + [3 + sumTo(2)]]
                  ^^^^^^^^
==> 5 + [4 + [3 + [2 + sumTo(1)]]]
                       ^^^^^^^^
==> 5 + [4 + [3 + [2 + [1 + sumTo(0)]]]]
                            ^^^^^^^^
==> 5 + [4 + [3 + [2 + [1 + 0]]]]
                        ^^^^^
==> 5 + [4 + [3 + [2 + 1]]]
                   ^^^^^
==> 5 + [4 + [3 + 3]]
              ^^^^^
==> 5 + [4 + 6]
         ^^^^^
==> 5 + 10
    ^^^^^^
==> 15
```

Tail Calls

Sum 1000000 0

```
sum 5 0
==> sum 5 0
==> sum 4 5
==> sum 3 9
==> sum 2 12
==> sum 1 14
==> sum 0 15
==> 15
```

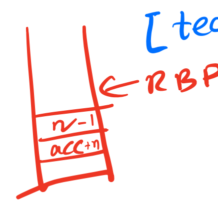
```
(defn (sum n acc)
  (if (= n 0)
      acc
      (sum (+ n -1) (+ acc n))))

(sum input 0)
```

```
e ::= n
    | true
    | false
    | input
    | x
    | (add1 e)
    | (let (x e1) e2)
    | (+ e1 e2)
    | (= e1 e2)
    | (if e1 e2 e3)
    | (set x e)
    | (block e1...en)
    | (loop e)
    | (break e)
    | (print e)
    | (call1 e)
    | (call2 e1 e2)
```

```
(defn (fac n acc)
  (if (= n 0)
    acc
    (if (= n 2)
      (* 2 (fac (+ n -1) (* acc n)))
      (fac (+ n -1) (* acc n))
    )
  )
)
```

start_sum:
  [setup stack]
  <BODY>
    push args
    call start_sum
  [teardown stack]

start_sum:
  [setup stack]
start_sum_body:
  <BODY>
    mov <old_args> <new_args>
    jump start_sum_body
  [teardown stack]



This is 64 bits: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

This is 5: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0101

This is 5 shifted 1 to the left, AKA 10: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1010

If we're OK with 63-bit numbers, can use LSB for *tag*

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1010 = 5

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 = false

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0011 = true

What does this mean for code generation?

What should we do the next time we need a new type? (string, heap-allocated object, etc.)

Condition Codes (that matter for us): Overflow, Sign, Zero

many instructions set these; arithmetic, shifting, etc. mov does not

cmp <reg>, <val> *compute <reg> - <val> and set condition codes (value in <reg> does not change)*

some cases to think about:

<reg> = -2^64, <val> = 1 Overflow: ___ Sign: ___ Zero: ___

<reg> = 0, <val> = 1Overflow: ___ Sign: ___ Zero: ___

<reg> = 1, <val> = 0Overflow: ___ Sign: ___ Zero: ___

<reg> = -1, <val> = -2 Overflow: ___ Sign: ___ Zero: ___

test <reg>, <val> *perform bitwise and on the two values, but don't change <reg>, and set condition codes as appropriate. Useful for mask checking.* test rax, 1 *will set Z to true if and only if the LSB is 1*

<label>: *set this line as a label for jumping to later*

jmp <label> *unconditionally jump to <label>*

jne <label> *jump to <label> if Zero is not set (last cmped values not equal)*

je <label> *jump to <label> if Zero is set (last cmped values are equal)*

jge <label> *jump to <label> if Overflow is the same as Sign (which corresponds to >= for last cmp)*

jle <label> *jump to <label> if Zero set or Overflow != Sign (which corresponds to <= for last cmp)*

shl <reg> *shift <reg> to the left by 1, filling in least-significant bit with zero*

sar <reg> *shift <reg> to the right by 1, filling in most-significant bit to preserve sign*

shr <reg> *shift <reg> to the right by 1, filling in most-significant bit with zero*